FIXED LENGTH MEMORY TO MEMORY INSTRUCTION SET

Inventors:
David A. Fotland
Tibet Mimar
Roger D. Arnold

RELATED APPLICATIONS

[0001] This application claims priority from U.S. provisional application number 60/213,745 filed on June 22, 2000, and from U.S. provisional application number 60/250,781 filed on December 1, 2000, which are both incorporated by reference herein in their entirety.

BACKGROUND OF THE INVENTION

1. Field of the Invention

[0002] The present invention relates to processor instruction sets, and more particularly to fixed length instruction sets for processors.

2. Description of Background Art

[0003] Over the past few years, Internet connectivity has been increasing at an astounding rate. Embedded processors are being increasingly used to provide this connectivity. Internet processing has little arithmetic, but involves manipulation of

memory buffers for packets and headers. A lot of data is moved without being transformed, especially in the communications field. Another often encountered task is to change or examine bit-fields in headers.

[0004] Conventionally, Reduced Instruction Set Computer (RISC) or Complex Instruction Set Computer (CISC) processors have been used for these tasks. However, both RISC and CISC prove to be problematic, especially when dealing with narrow data (e.g., 16-bit data) which is common in the communications field.

[0005] When data in memory needs to be manipulated, RISC processors traditionally move data from memory to registers ("Load"), perform the desired arithmetic calculations on them, and then move the result back from the register to memory ("Store"). The Arithmetic Logic Unit (ALU) instructions and the data from the registers are 32-bit wide in RISC processors. The data from memory may be of 8, 16, or 32 bits. If the data is 8 or 16 bits wide, it is sign extended to make it 32 bits before arithmetic is performed on it. Once the arithmetic is performed, it is truncated to 8 or 16 bits and then stored in memory. Due to the Load & Store architecture of RISC processors, numerous instructions are often required for simple operations. In addition, one RISC instruction can use at most one memory operand (e.g., load and store instructions), or no memory operands (e.g., arithmetic instructions).

[0006] In CISC processors, operations can be performed on data in memory without having to load and then later store the data. The instruction set itself specifies the width of the operands, as well as the width of the arithmetic. However, in order to do this, the instruction set in CISC processors is variable in length.

[0007] Further, although many conventional chips include Direct Memory Access (DMA) engines to offload data movement from the main processor and special purpose assists, these DMA engines or special purpose assists take up valuable silicon real-estate, and make the software more complex.

[0008] In the communications field, narrow data (e.g., 16-bit data or 8-bit data) needs to be dealt with efficiently. In particular, communications data is often 16 bits wide. Therefore similar support for 32-bit wide data and 16-bit wide data is needed. That is, a processor should be similarly robust for processing 32-bit wide data as well as 16-bit data. At the same time, keeping the arithmetic and register data at 32 bits simplifies the hardware design. However, narrow data in 32-bit registers can be problematic. One of the reasons why 16-bit data in 32 bit registers complicates hardware is because it can result in partial register writes. A partial register write occurs as a result of an instruction which changes part of a register and leaves the rest of the register unmodified. In a pipelined machine, the bypassing of data from one instruction to the next becomes much more complicated because of partial writes. Further, instructions may need to be specific regarding which part of a 32-bit register to use.

[0009] In conventional systems, memory operand width is dependent on ALU width. For instance, if the ALU is 32 bits wide, the memory operand is also 32 bits wide. In contrast, if the memory operand width were to be independent of the ALU width, data could be stored more efficiently in memory. For instance, if the memory operand were to be, say, 16 bits wide, it could be stored in memory as 16-bit data, rather than extending it to 32 bits.

[0010] Therefore, what is needed is (1) a processor with memory operand widths which are independent of the ALU width; (2) a processor to provide similarly robust processing for 32-bit and 16-bit data, without significantly increasing the complexity of hardware and software; and (3) fixed-length instructions which can have multiple memory operands.

SUMMARY OF THE INVENTION

[0011] The present invention is a method and system for a fixed length memory-to-memory instruction set. The present invention provides similarly robust support for 16-bit and 32-bit data. Further, the present invention is a method and system for implementing a memory operand width independent of the ALU width.

[0012] A fixed length instruction set allows for fast pipeline processing. At the same time, the ability to access data from memory directly without first having to load it to a register is important. The instruction set in accordance with one embodiment of the present invention comprises fixed length instructions, and allows direct access to memory. Thus a system in accordance with the present invention enables very high performance when processing network traffic. The instruction set is small and simple, so the implementation is lower cost than traditional processors. Since the instruction set is more efficient, fewer instructions are required, so less memory is used, which further reduces cost. Moreover, the processor can operate at a lower clock frequency, saving power.

[0013] In one embodiment of the present invention, general instructions (e.g., add) can have a first source operand from memory, a second source operand from a register, and a memory operand as a destination. Like a RISC processor, the arithmetic and the register operand are 32 bits. However, in the present invention, the size of the memory operands is independent of the ALU width. The instruction specifies the size of the memory operands. Having the instruction specify the operand width for memory access saves encoding space in the instruction set. Further, the memory operands are encoded using a fewer number of bits, thus enabling a 32-bit instruction with multiple memory operand specifiers.

[0014] A processor's ability to handle two memory operands in one instruction reduces the number of instructions required to perform many functions. For instance, data can be moved from one memory location to another in a single instruction. Another example of the use of two memory operands is that data from memory can be added with data from a register and stored in memory with a single instruction. This function can be performed without the steps of loading the data to a register, performing the add, and then storing the result. The present invention serves to eliminate several of the load and store instructions inherent in RISC architecture, while still maintaining simple hardware and software.

[0015] In addition to using multiple memory operands in a single instruction, in one embodiment of the present invention, each of the memory operand specifiers are provided with more powerful addressing modes. These addressing modes permit a higher code density, and thus a relatively smaller code size.

[0016] In an aspect of the present invention, a system in accordance with one embodiment of the present invention can implement a single-bit semaphore. In another aspect of the present invention, a "shift-and-merge" instruction can be implemented to access information which is split across two words. In yet another aspect of the present invention, in a multi-thread environment, instruction read, instruction write and instruction erase instructions are implemented using two slots assigned to a specific thread in order to avoid interference with other threads.

BRIEF DESCRIPTION OF THE DRAWINGS

[0017] Figure 1A is a block diagram representing a RISC processor.

[0018] Figure 1B is a flowchart illustrating the steps performed by a RISC processor.

[0019] Figure 1C is an example of an instruction format used by a RISC processor.

[0020] Figure 1D is another example of an instruction format used by a RISC processor.

[0021] Figure 2 is a block diagram representing a CISC processor.

[0022] Figure 3A is a block diagram of a processor in accordance with an embodiment of the present invention.

[0023] Figure 3B is a flowchart illustrating the steps performed by a processor in accordance with one embodiment of the present invention.

[0024] Figure 4 is an instruction format for an embodiment of the present invention.

[0025] Figure 5 illustrates how a memory operand address is calculated in one embodiment of the present invention.

[0026] Figure 6 illustrates the encoding of various addressing modes in one embodiment of the present invention.

[0027] Figure 7 illustrates the implementation of a single bit semaphore.

[0028] Figure 8 illustrates a shift-and-merge instruction.

[0029] Figure 9 illustrates time slots assigned to multiple threads.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0030] A preferred embodiment of the present invention is now described with reference to the figures where like reference numbers indicate identical or functionally similar elements. Also in the figures, the left most digit of each reference number corresponds to the figure in which the reference number is first used.

[0031] One type of conventional processor has a Reduced Instruction Set Computer (RISC) architecture. Figure 1A is a block diagram illustrating one possible implementation of a conventional RISC processor 100. The RISC processor comprises a memory 110, a register file 120, and an Arithmetic Logical Unit (ALU) 130. The RISC processor 100 may also include a sign extender 115 and a truncator 125. The register file 120 can include one or more registers.

[0032] Figure 1B is a flowchart illustrating the processes performed by processor 100. It includes the steps of loading 132 data to the register file 120, performing 134 computations on the data, and storing 136 data back to memory 110. As can be seen from Figure 1B, the RISC processor 100 has a "load and store" architecture. Data from memory 110 cannot be accessed directly. The data from memory 110 is first loaded 132 into the register file 120. The ALU 130 only obtains data from the register file 120. The width of the arithmetic in the ALU 130 is 32 bits. The width of data from the register file 120 is also 32 bits. The width of the data from the memory 110 may be variable. If the width of the data from the memory 110 is 8 or 16 bits, the sign extender 115 is used to extend the data to 32 bits before it is stored in the register file 120. The ALU 130 then performs 134 the requisite computations on the data obtained from the register file 120, and the result is stored in the register file. This result can then be stored 136 from the register file 120 back to memory 110. When the result is stored 136 in memory, a truncator 125 may be used to store the result as 8-bit or 16-bit data. Alternately, the result may be stored as a 32-bit data in the memory 110.

[0033] Figure 1C and 1D illustrate two possible formats for two fixed-length instructions used by the RISC processor 100. In each case, the instruction is 32 bits in length. Figure 1C illustrates an instruction with three operands – two sources and one destination. An arithmetic instruction could have such a format. In Figure 1C, 6 bits are used to identify the operation to be performed (the Operation Code or "OPCode"). 5 bits are used to encode the address of the first source S1, 5 bits are used to encode the second source S2, and 5 bits are used to encode the address of the target T. The remaining 11

bits are used to encode a sub-OPCode. This sub-OPCode can be used in conjunction with the OPCode, to specify one of a larger number of operations. Figure 1D illustrates an instruction with one source data, an offset (or immediate), and a target. A "load" instruction could have such a format. In such a case, 6 bits are used to identify the OPCode, and 5 bits each are used to encode the address of the source and the target. The remaining 16 bits are used for the offset (or immediate).

[0034] Another type of conventional processor is one with a CISC architecture. Figure 2 is a block diagram illustrating one possible implementation of a CISC processor 200. The CISC processor 200 comprises a memory 110, a register file 120, an ALU 130, and a multiplexor 240.

[0035] The CISC processor 200 does not have a load-store architecture. Operands in memory 110 can be accessed directly by the ALU 130. The CISC processor 200 can perform arithmetic on two operands. One of these operands can be either from memory 110 or from register file 120. The second operand is from register file 120. The multiplexor 240 selects whether the first operand will be from memory 110 or from register file 120 for a given operation. Once the requisite arithmetic is performed by the ALU 130, the result can be stored either in the memory 110 or in the register file 120.

[0036] In the case of a CISC processor 200, the width of the operands remains the same all the way from the memory 110 to the ALU 130 and back to memory 110. The instructions in a CISC processor themselves specify the width of the add. A CISC processor 200 however, has variable length instructions. Variable length instructions

add complexity to the system. One of the reasons for this is that it is more computation-intensive for processors to fetch and decode variable length instructions.

[0037] Figure 3A is a block diagram of one embodiment of the present invention. The processor 300 comprises a memory 110, a register file 120, an ALU 130, a multiplexor 240, an immediate 302, a sign extender 115, and a truncator 125.

[0038] In one embodiment of the present invention, the ALU 130 performs arithmetic on two operands. The first source operand is a general operand. It can be from the register file 120. Alternatively, it can be an immediate 302. If the immediate is not 32 bits wide, it is passed through the sign extender 115 to make it 32 bits wide. Further, the first operand can also be from memory 110. If the data stored in memory 110 is 8-bit or 16-bit data, it is extended by the sign extender 115 up to 32 bits in one embodiment of the present invention. The second source operand is from the register file 120. The ALU 130 then performs the requisite arithmetic on the 32-bit wide data. The result can then be stored either in register file 120, or directly in memory 110. If the result is stored directly in memory 110, a truncator 125 may be used to store the result as 8-bit or 16-bit data.

[0039] Like a RISC processor 100, the processor 300 has a 32-bit ALU, and 32-bit data coming from the register file 120. Unlike the RISC processor 100, however, the processor 300 can access data from, and store data in, memory 110 directly, and does not have a "load and store" type of architecture. The data from memory 110 can be of any size, and this size is specified by the instruction. Instructions used by processor 300 are further described with reference to Figure 4 and Figure 6.

[0040] Like the CISC processor 200, the processor 300 has one operand coming from the register file 120, while the other operand can come from either the memory 110 or the register file 120. However, unlike the CISC processor 200, the processor 300 does not have variable length instructions.

[0041] Figure 3B is a block diagram which illustrates the various stages in the pipeline which processor 300 employs in one embodiment of the present invention. In the embodiment illustrated in Figure 3B, the first operand is from memory 110. Each of the steps depicted in Figure 3B occurs in one clock cycle, with the next step occurring in the next cycle. In the first clock cycle, the processor 300 fetches 312 an instruction. In one embodiment of the present invention, this instruction specifies the size of the memory operand. In one embodiment, the size of the memory operand is specified in the Operation Code (OpCode). For a further discussion of a format of the instruction, refer to the discussion regarding Figure 4 below. Having the instruction specify the operand width for memory access saves encoding space in the instruction set. The smaller encoding enables a 32-bit instruction with two memory operand specifiers.

[0042] In the second clock cycle, the memory operand address is then calculated 314. The details regarding the calculation of the memory operand address are discussed with reference to Figure 5. Also in the second clock cycle, data is read 316 from the register file 120.

[0043] In the third clock cycle, data is read 318 from the memory location specified by the memory operand address calculated 314. In the next clock cycle, arithmetic is performed 320 by the ALU 130. In the fifth clock cycle, the result of the

operation is then directed 322 to the destination. The destination for the result can be memory 110, a register file 120, or, in some cases, an immediate.

[0044] When an immediate is specified as a destination for an operation, functionally, this indicates that the result of that operation is stored nowhere. Nonetheless, the operation is actually performed, and this can result in valuable "side-effects." For instance, an address register can be auto-incremented in this manner. With reference to Figure 3B, it can be seen that by using the immediate as a destination, an address register can be incremented in clock cycle 2 itself, rather than waiting until clock cycle 5. Thus the auto-incrementing can occur much faster in this manner. Another instance where the immediate can be used as a destination is for setting a condition code. A condition code can be used, for example, to compare A and B, and determine which branch of a tree to follow based on which of A and B is greater. This comparison can be performed by performing the operation of subtracting B from A, and using the result to set the condition code. However, there is no value in actually storing this result. In such a situation, the result, once known, can be "thrown away" by specifying an immediate as the destination for the operation.

[0045] In a conventional system a typical set of condition codes includes four condition codes (negative, zero, overflow, and carry). In contrast, a processor in accordance with one embodiment of the present invention has a set of eight condition codes: the above-mentioned four codes for 16 bits, and the above-mentioned four codes for 32 bits. The conditional brand instructions can select either the 16-bit codes or the 32-bit codes.

[0046] In a system in accordance with one embodiment of the present invention, a Load Effective Address (LEA) operation is performed. As can be seen from Figure 3B, in general, the memory address is calculated 314, the data from the memory is then read 318, and the result is stored 320. (This can also be seen with reference to Figure 3B). However, it is sometimes useful to simply calculate 314 an address, and to store 320 the result, without reading 318 the data from memory. As discussed above, each step illustrated in Figure 3B occurs in one clock cycle. Thus if the steps in Figure 3B are performed, it takes several clock cycles for the result to be stored 320. Specifically, the result is stored 320 in clock cycle 5. Instead, LEA occurs within clock cycle 2.

[0047] Figure 4 illustrates one instruction format 400 used by processor 300 in one embodiment of the present invention. In one embodiment, processor 300 uses fixed-length instructions that are 32 bits wide. In alternate embodiments, the fixed-length instructions may have a different size. The instruction format 400 in Figure 4 has three operands – two sources, and one destination. In an embodiment, the processor may use different instruction formats, such as those operating with one or two operands, and those using immediates.

[0048] In instruction format 400, out of the 32 available bits, 5 bits are used to represent the Operation Code (OPCode). 11 bits are used to encode the destination, as well as one source. 5 bits are used to represent the second source. The 5-bit source has to be a register 110. Specifically, since 5 bits are used to encode this source, this operand data has to be from one of 32 registers. The other source and the destination are each encoded using 11 bits. This source and destination can each be from either memory

110 or register file 120. In other words, by using this instruction format 400, processor 300 can have two operands (one source and the destination) from memory 110, while still having fixed-length instructions. In one embodiment of the present invention, it is possible to have two memory operands in a single instruction because a memory operand can be encoded using only 11 bits. The manner in which a memory operand can be encoded using 11 bits is discussed in detail below with reference to Figure 6.

[0049] A processor's ability to process two memory operands in one instruction reduces the number of instructions required to perform some functions. For instance, by using a "move" operation, data can be moved from one memory location to another in a single instruction. In a conventional RISC processor 100, this would involve loading data from the first memory location to a register, and then storing the data from the register to the second memory location. Thus, multiple instructions would be required by a conventional RISC processor 100 to perform move data from one memory location to another memory location. Table 1 compares moving data from location A in memory to location B in memory with a conventional RISC processor and with a system in accordance with an embodiment of the present invention.

Steps	Instructions with a conventional RISC processor	Instructions with an embodiment of the present invention
1	Load A	Move B, A
2	Store B	

Table 1

[0050] Another example of the use of two memory operands is that data from memory can be added with data from a register and stored in memory. The first source operand could identify the memory location for the data to be added, the second source operand could identify the register, and the destination could identify the memory location where the result is to be stored. Thus, by using a system in accordance with the present invention, this operation can also be performed with a single instruction. Unlike in a conventional RISC processor 100, this operation can be performed without the steps of loading the data to a register, performing the add, and then storing the result. Table 2 compares the computation of C = A + B (where A and C are memory locations) with a conventional RISC processor and with a system in accordance with an embodiment of the present invention.

Steps	Instructions with a conventional RISC processor	Instructions with an embodiment of the present invention
1	Load A	Add C, B, A
2	Add C, B, A	
3	Store C	

Table 2

[0051] From the above examples, it can be seen that the present invention serves to eliminate several of the load and store instructions inherent in RISC architecture.

[0052] Referring again to Figure 4, the instruction format 400 specifies the size of the memory operand as mentioned above. In one embodiment, the size of the memory

operand is specified in the OPCode. In one embodiment of the present invention, one, two, and four are use for 8-bit, 16-bit, and 32-bit operands respectively. For example:

Move.1 implies Move one byte;

Move.2 implies Move two bytes;

Move.4 implies Move four bytes.

[0053] However, not all instructions support all three operand sizes. For example, the multiply instruction will not use any suffixes, as only multiplication with 2 bytes is permissible in one embodiment. Use of Mul.2 in such a case may lead one to think that Mul.1 and Mul.4 are also allowed. Therefore, only instructions that allow multiple operand sized have suffixes, and some instruction may only support 16-bit and 32-bit sizes.

[0054] Figure 5 is a block diagram illustrating how, in one embodiment of the present invention, the memory operand address is calculated 314. Figure 5 illustrates several components of processor 300 including an address register file 512, a general register file 514, an adder 516, and a multiplexor 240. In one embodiment of the present invention, processor 300 has 8 address registers in the address register file 512 and 32 general registers in the general register file 514. A distinction is made between address registers and general registers, so as to achieve some of the addressing modes discussed below.

[0055] The location of the first operand (discussed earlier with reference to Figure 3A) can be calculated using various address modes. In one embodiment, the present invention has the addressing modes discussed below. These powerful addressing

modes for the first operand permit a higher code density, and thus a relatively smaller code size.

[0056] The first three addressing modes are illustrated in Figure 5, and the other two are illustrated in Figure 3A:

[0057] 1. Register + Immediate Addressing Mode:

In this addressing mode, the value in the register added to the immediate gives the memory address in which the memory operand is located. In one embodiment of the present invention, the register + immediate addressing mode is implemented as Address Register + 7-bit immediate. The term "address register" is further discussed below with reference to Figure 6.

[0058] 2. Register + Register Indirect Addressing Mode:

In one embodiment of the present invention, this addressing mode is implemented as Address Register + General Register. These terms are discussed further with reference to Figure 6. The value in the Address Register added to the value in the General Register gives the memory address in which the memory operand is located.

[0059] 3. Register + Immediate Auto-Increment Addressing Mode:

In one embodiment, this is implemented as auto-increment in conjunction with the Address Register + 4-bit immediate addressing mode. In one embodiment of the present invention, there are two different implementations of this addressing mode:

[0060] (i) Pre-increment: In this mode, the Address Register itself is used for the memory address in which the operand data is located, and the Address Register + 4-bit Immediate is the new value stored in the Address Register.

[0061] (ii). Post-increment: In this mode, Address Register + 4-bit Immediate is used for both the memory address in which the operand data is located, and for storing a new value in the Address Register.

[0062] 4. Register Direct Addressing Mode:

In this addressing mode, the data in the register itself is used as the first operand.

[0063] 5. Immediate Addressing Mode:

In this addressing mode, the value of the immediate itself is used as the first operand.

[0064] It can be seen that in one embodiment, the present invention does not inherently have a direct memory addressing mode. (In such an address mode, the address of the memory in which the first operand is located is directly specified.) However, in one embodiment of the present invention, however, such a direct memory addressing mode can be used by inserting an immediate into an address register. This is done using the Move Address register Immediate (MoveAI) instruction. In particular, by using the MoveAI instruction in conjunction with the Address Register + 7-bit Immediate instruction, a large number of bits can be used to specify the memory address. In one embodiment, a 24-bit Immediate can be moved into the Address Register. This

address register can then be used in conjunction with the Address Register + 7-bit Immediate instruction, to obtain a 31-bit memory address.

[0065] Figure 6 illustrates how, in one embodiment of the present invention, the first operand is encoded using 11 bits, and how the various addressing modes discussed above are encoded. Each of the rows in Figure 6 corresponds to one of the addressing modes discussed above.

[0066] Row 1 corresponds to the Address Register + 7-bit immediate mode. The 7-bit immediate is represented by I_0-I_6. In one embodiment of the present invention, 3 bits (A_0–A_3) are used to specify the address register 512. Since only 3 bits are used to specify the address register, one of only 8 address registers can be specified. In Row 1, the leftmost "1" in the row indicates that the Address Register + 7-bit immediate mode is being used.

[0067] Row 2 in Figure 6 corresponds to the Register + Register addressing mode discussed above. One of these registers is selected from the 8 address registers in the address register file 512, while the other is selected from the 32 data registers in the data register file 514.

[0068] Row 3 corresponds to the Address Register + 4-bit addressing mode with auto-increment. The 4-bit immediate is represented by I_0-I_3. The Address Registers are encoded by bits A_0–A_3 as above. In one embodiment, the "M" indicates the post (if M=0) or pre (if M = 1) addition of the increment mode.

[0069] Row 4 corresponds to the Register Direct addressing mode discussed above. Conventional systems typically have 5 bits to specify registers as discussed

above with reference to Figures 1C & 1D. Therefore only 32 registers can be directly addressed. Thus, most conventional processors have separate addressing modes for general registers and control registers. In contrast, as indicated in Row 4, the present invention has 8 bits for specifying registers. Thus 256 registers can be directly addressed in a system in accordance with an embodiment of the present invention. Hence any instruction in an embodiment of the present invention can operate on any register, regardless of whether the register is a general register, address register, or control register.

[0070] Row 5 corresponds to the Immediate addressing mode discussed above. In this case, the 8-bit immediate is used as the first operand.

[0071] In Rows 2 through 5, the three leftmost bits are used to indicate which addressing mode is being used.

[0072] The various addressing modes for the present invention discussed above effectively allow, in one embodiment, all of the on-chip Static Random Access Memory (SRAM) to be treated as a very large register file. That is, the data memory locations can be accessed in one cycle, the same as registers, and they can be used as source or destination operands in most instructions.

[0073] Figure 7 comprises an ALU 130, a source 710, and a mask 720. The source 710 is a general source. In one embodiment, it can be from memory. In another embodiment, it can be from a register file. In yet another embodiment, it can be an immediate.

[0074] Figure 7 illustrates how, in accordance with one aspect of the present invention, a single bit semaphore is implemented. This is in contrast to some conventional systems which use more bits (e.g., an entire 32-bit word) to implement a semaphore. Semaphores are needed when an instruction set can support multiple processors, or a single processor with multiple threads. A semaphore is a hardware or software flag. In multitasking systems, a semaphore is a variable with a value that indicates the status of a common resource. A semaphore can be used to lock the resource that is being used. A process needing the common resource checks the semaphore to determine the resource's status and then decides how to proceed.

[0075] In one embodiment, the bit number to be set or cleared is specified in the instruction. Based on this bit number, a mask 720 is created. If the bit is to be set, the mask 720 has all zeros, except for a 1 in the position of the bit to be set. This mask is then ORed with the source 710 data to set the bit. If the bit is to be cleared, the mask 720 has all 1s, except for a zero in the position of the bit to be set. This mask is then ANDed with the source 710 data to clear the bit. The prior value of the bit (before it is set or cleared) is used to set a condition code. Thus the present invention implements a single bit semaphore.

[0076] In another aspect of the present invention, shift-and-merge instructions are used to access data across word boundaries. Network data enters the processor as a stream of bytes. Sometimes, the data of interest falls across two consecutive "words." A word is typically comprised of 32 bits. An example of this is shown in Figure 8. The data of interest is AB, which falls across Word 1 and Word 2.

[0077] Memory accesses typically access one byte at a time. Therefore the situation described above will require two memory accesses – one to access byte A, and another to access byte B. Moreover, these two bytes (A & B) then have to be put together in order to procure the data of interest.

[0078] Various architectures solve this problem in different ways. For instance, a RISC instruction set may use "Load Left" and "Load Right" instructions. A RISC processor is generally used to manipulate 32-bit data. "Load Left" and "Load Right" instructions imply that data from Word 1 is written to the left half of a register, and data from Word 2 is written to the right half of the register in order to obtain the data desired. However, this results in partial register writes, which slow down pipeline processing.

[0079] In one embodiment of the present invention, a "shift-and-merge" instruction is used to circumvent this problem. The present invention is often used to manipulate 16-bit data. Instead of pulling out the whole of both Word 1 and Word 2, only the byte of interest is pulled from each word. In this case, byte A is loaded into a register. Then a "shift-and-merge" instruction is executed, which shifts data (in this example, byte A) to the left by one byte, and merges new data (in this example, byte B) into the same register. This is illustrated in Figure 8. In alternate embodiments of the present invention, more than 1 byte may be shifted-and-merged.

[0080] In yet another aspect of the present invention, Instruction Read/Instruction Write/Instruction Erase (IWRITE/IREAD/IERASE) instructions are used in multi-thread environments. Most modern processors have a single memory, which is used for storing both instructions and data. However, in one embodiment of the

present invention, the processor has separate memory spaces for data and instructions. There are several reasons for this separation of the data and instruction memories. First, since instructions in accordance with the present invention allow for two memory operands in a single instruction, data memory needs to have two ports. Instruction memory, on the other hand, need have only one port. If a single memory space were used for instructions and data, two ports would have to be implemented for all of it. Since two port memories require more silicon real-estate than one port memories, bifurcating the memory space into data memory and instruction memory makes for a smaller chip. Second, execution of instructions should be deterministic. That is, even though multi-threading may be permitted, the performance of one thread should not affect the performance of another thread. If a single memory space existed for data and instructions, modifying instructions while data is being fetched may cause problems.

[0081] For at least these reasons, in one embodiment of the present invention, memory is separated out into data memory and instruction memory. It is thus needed to read, write and erase not only the data memory, but also the instruction memory. The IREAD, IWRITE, and IERASE instructions are designed for this. In a multi-thread environment, it is important for these instructions to be non-interfering with other threads.

[0082] Figure 9 illustrates a multi-thread environment. The X-axis represents multiple thread, T1, T2, and T3. The Y-axis represents time. Each thread T1, T2, and T3, has its own slots in time, which do not interfere with the time slots assigned to any of the other two threads. Thread T1 has Slot1,1 – Slot1,5. Thread T2 has Slot2,1 –

Slot2,3. Thread T3 has Slot3,1 – Slot3,2. In general, one instruction is performed per slot. However, when an IREAD, IWRITE, or IERASE instruction is encountered, the thread actually accesses the instruction to be read, written, or erased in the next slot assigned to it. For instance, if the instruction in Slot2,1 were an IREAD, Slot2,2 would actually access the instruction to be read. The next instruction in T2 would be implemented only in Slot2,3. Using two slots (instead of one) for each of these instructions ensures that IREAD, IWRITE, and IERASE are non-interfering with other threads.

[0083] While particular embodiments and applications of the present invention have been illustrated and described, it is to be understood that the invention is not limited to the precise construction and components disclosed herein and that various modifications, changes and variations which will be apparent to those skilled in the art may be made in the arrangement, operation and details of the method and apparatus of the present invention disclosed herein without departing from the spirit and scope of the invention as defined in the following claims.